什么是彩虹表？

说起彩虹表有必要提一下 hash 函数，hash 函数又叫散列函数，对于任意 hash 函数应具备以下特点。

1. 压缩性：对于任意给定输入有唯一特定长度输出，例 SHA1 的 hash 值为 20 字节。
2. 容易计算：即从原始数据计算 hash 值应该很容易。
3. 抗修改：对原始数据哪怕 1bit 的修改都会对 hash 值产生重大影响。
4. 抗碰撞：这里分为强碰撞和弱碰撞，弱碰撞是给定一个数据很难找出相同 hash 值得另外一个数据。强碰撞是很难找出 hash 值相同的两个数据。

那么什么地方用到了 hash 函数呢？很久以前，有些网站对用户的密码采用的是明文存储，用户登录时，服务器根据用户输入的用户名和明码和数据库里的用户名密码进行比对，匹配成功则成功登录。很显然这样做存在极大的安全隐患，在早期 https 还没有大规模使用时，用户数据是通过明文的方式在网络里传输的，这样用户的密码很容易被 hacker 窃取。因此，有些注重安全的网站尝试只存储用户密码的 hash 值，然后根据用户输入的密码的 hash 值和数据库里的 hash 值作比较，一定程度上提高了安全性。这是利用 hash 函数的单向性，即根据 hash 值推断出输入值是很难的（PS：hash 函数还广泛的用在校验中，如当你从某网站下载软件时，一般都会提供软件的数字签名，以防恶意人员篡改软件植入木马病毒。程序员世界里很火的 git 的 commit 号也是一个 hash 值）。

聪明的黑客想到了用预计算的哈希链（Precomputed hash chains）的方法来破解登录密码。对于特定要破解的 hash 函数，首先根据输入域定义一个 R 函数（Reduction function），该函数的定义域和值域需要和 hash 函数相反，即该函数的定义域是 hash 函数的值域，值域是合理的输入域（例如某网站要求密码是 6 位数字组合），则 R 函数的值域就是所有可能的 6 位数字组合。通过该函数可以将 hash 值约简为一个合理的登录密码。这里需要强调的是，由于 hash 函数单向的，所以 R 函数并非是 hash 函数的反函数。举个简单的例子，例如 6 位数字明文“123456”进行 H 运算后得到了 D2A82C9A，而对 D2A82C9A 进行 R 运算后得到另一个五位字母格式的值“446231”。因为这个值落在 H 的定义域中，因此可以对它继续进行 H 运算。就这样，将 H 运算、R 运算、H 运算……这个过程反复地重复下去，重复一个特定的次数 k 以后，得到的 hash 值为 A293BC11 就得到一条哈希链，这条链条并不需要完整地保存下来，只需要保存其头结点 123456 和尾节点 A293BC11。以大量的随机明文作为起节

点，通过上述步骤计算出哈希链并将终节点进行储存，即可得到一张哈希链表。这张链表需要如何使用呢？

对于一个给定的 hash 值，首先进行 R 运算。然后和链表的尾节点进行比较，如果没有找到，则进行 H 和 R 运算，然后继续比较。直到和某个尾节点相同。然后根据尾节点找到头结点，从头结点开始做 H、R 交替运算，直到某个 hash 值和给定 hash 值相等，或者找到尾节点。

这里为什么不是肯定能找到和给定 hash 值相等的 hash 呢？

因为在构造哈希链的时候，一个优秀的函数 R 功不可没。首先 R 需要能将值域限定在固定的范围，例如给定的长度范围、给定的字符取值范围之内，否则的话，哈希链中大量的计算结果并不在可接受的取值范围内，一条链条无法对应多个明文，链条就失去了意义；其次 R 必须同哈希函数一样，尽量保证输出值在值域中的均匀分布，减少碰撞的概率。然而实际上，很难找到能满足这些需求的完美的 R 函数。当计算中发生碰撞时，两条链表后面的节点将会完全重复。这样两条哈希链所包含的明文数量就远小于理论上的明文数 2×k。可怕的是，由于集合只保存链条的首末节点，因此这样的重复链条并不能被迅速地发现。随着碰撞的增加，这样的重复链条会逐渐造成严重的冗余和浪费。

彩虹表的出现就是为了解决这个问题。

2003 年提出的彩虹表算法进行了针对性的改进。它在各步的运算中，并不使用统一的 R 函数，而是分别使用 R1…Rk 共 k 个不同的 R 函数。这样生成的哈希链集即被称为彩虹表。（在不同的运算位置使用不同的 R 函数，就像彩虹由内而外的不同位置上显示出不同的颜色一样。）这样一来，如果发生碰撞，不难发现，当两个链条发生碰撞的位置并非相同的序列位置时，后续的 R 函数的不一致使得链条的后续部分也不相同，从而最大程度地减小了链条中的重复节点，保证了链条的有效性。同时，如果在极端情况下，两个链条有 1/k 的概率在同一序列位置上发生碰撞，导致后续链条完全一致，这样的链条也会因为末节点相同而检测出来，可以丢弃其中一条而不浪费存储空间。彩虹表的使用比哈希链集稍微麻烦一些。首先，假设要破解的密文位于任一链条的 k-1 位置处，对其进行 Rk 运算，看是否能够在末节点中找到对应的值。如果找到，则可以如前所述，使用起节点验证其正确性。否则，需要继续假设密文位于 k-2 位置处，这时就需要进行 Rk-1、H、Rk 两步运算，然后在末节点中查找结果。如是反复，最不利条件下需要将密文进行完整的 R1、H、…Rk 运算后，才能得知密文是否存在于彩虹表之中。

对于相同个数的明文，当 k 越大时，破解的期望时间就越长，但彩虹表所占用的空间就

越小；相反，k 越小时，彩虹表本身就越大，相应的破解时间就越短。这正是保持空间、时间二者平衡的精髓所在。极端的，令 k=1，简化函数 R(x)=x，这样的彩虹表就变成了通常的错误理解，即将明文、密文对应关系全部保存的表。此时由于 k 极小，因而得到的表的体积极大，甚至可能超出储存能力。（当然，对于范围较小的明文，如 6 位以下数字英文的组合，或是全世界常用密码的集合，生成 k=1 的表还是不费什么事的。）

再谈 R 函数

那么如何防御彩虹表呢，聪敏的安全人员早就想到了很好的彩虹表防御办法。

最常用的方法就是加盐（salt），那么什么是加盐呢？在密码学中，加盐是指通过在密码任意固定位置插入特定的字符串，让散列后的结果和使用原始密码的散列结果不相符，这种过程称之为加盐。因此即使通过彩虹表找到了特定 hash 值对应的密码，也无法得到用户输入的密码，甚至根本得不到密码。加入破解者知道盐值和插入位置，也需要针对加盐的 hash 函数做对应的 R 函数修改，因此已有的彩虹表数据就完全无法使用，必须针对特定的 H 重新生成，这样就提高了破解的难度。

防御彩虹表的另一种方法是提高 hahs 函数的计算难度，提高单次 hash 的计算难度有违 hash 的设计初衷。因此常见的方法是对特定的密码做大量反复的 hash 操作。例如将 H 定义为计算一千次 MD5 后的结果。这样的话，每个彩虹表项将对应 1000 个 hash 值和一个合法密码，这样的话彩虹表的 k 值不能取得太大，否则无法再可接受的时间内完成破解，而如果将 k 值取值减小，又大大的增加了彩虹表的空间。从而提高破解的成本。